Term-graph rewriting in Tom using relative positions

Emilie Balland and Paul Brauner

UHP & LORIA, INPL & LORIA
Campus Scientifique, BP 239,
54506 Vandœuvre-lès-Nancy Cedex France

Abstract

In this paper, we present the implementation in Tom of a de Bruijn indices generalization allowing the representation of term-graphs over an algebraic signature. By adding pattern matching and traversal controls to Java, Tom is a well-suited environment for defining program transformations or analyses. As some analyses, e.g. based on control flow, require graph-like structures, the use of this formalism is a natural way of expressing them by graph rewriting.

1 Introduction

Program transformation and graph rewriting are strongly related [9]. Indeed, although the structure of a program may be represented by a tree, informations about its execution like data dependencies or control flow are naturally expressed by data-structures inherently using cycles or subterms sharing, in other words by graphs. More precisely, since these graphs are oriented and labelled over an algebraic signature, such transformations are described within the framework of term-graphs [12]. There exists several definitions of term graph rewriting, category-theory oriented [6,10], equationally oriented [1] or implementation-oriented [2].

Since 2001, the Protheo team is developing the Tom system [11], whose main originality is to be built on top of an existing language Java. Tom provides pattern matching facilities to inspect objects and retrieve values. Moreover, the rewriting steps can be controlled using a powerful strategy language. The main application of the language being program transformation and code analysis, we were interested in extending the Tom language for supporting term-graph transformations.

In this paper, we introduce the notion of relative position inspired from the de Bruijn indices as a way to express paths between two subterms. Then we present an implementation of term-graphs based on this formalism. As Tom

provides rewriting strategies, integrating such structures in the language offers strategic graph rewriting for free. After introducing the notion of relative positions, we will explain how the language can be extended to offer facilities for strategic graph rewriting. Finally, we will illustrate the use of this extension by an implementation of lambda-calculus normalization.

2 Term-graph representation

Our goal is to represent term-graphs on top of the term rewriting theory with the fewer possible modifications to this formalism to take advantage of the existing result and tools, namely Tom. The main idea of this paper is to raise the notion of position to the level of first-order terms by extending algebraic signatures with an infinite set of constants representing positions. This allows for the description of terms containing some "pointers" to subterms of themselves. As an example, the term $s(a, 1)$ defined over such a signature denotes a term whose second child references the first-one.

The main issue of this representation is that it is context sensitive. For instance, the position 1.2 references the subterm a in $f(s(1.2, a))$, but $s(1.2, a)$ in $f(f(s(1.2, a)))$. This raises the idea of *relative positions* describing paths inside a term to the referenced subterms. The previous example would then be written $f(s(a, -1.1))$, where -1 indicates one backward step inside the term. This can be seen as a generalization of de Bruijn indices extended to the count of all function symbols, not only abstractions.

In this section, we define more formally this notion of relative position and terms with references before we present an implementation aimed to be used by Tom. We finally discuss the relation between this formalism and term-graphs as well as the associated technical solution.

2.1 Terms with references

As usual, a *position* is a finite sequence of natural numbers. The subterm u of a term t at position ω is denoted $t_{|\omega}$, where ω describes the path from the root of t to the root of u. To emphasize the difference with relative positions, we will sometimes refer to positions as *absolute positions*.

Let us first define relative positions along with their meaning.

Definition 2.1 (Relative position) *The set Rpos of relative positions is the monoid $(\mathbb{Z}^*, .)$ with neutral element Λ where $\mathbb{Z}^* = \mathbb{Z} \setminus \{0\}$.*

We note n, p the elements of $\mathbb{Z}^*$ and ω_r, ω'_r, ... the elements of *Rpos*.

Definition 2.2 (Referenced subterm) *Given an absolute position ω and a relative position ω_r, the absolute position accessed by ω_r from ω is written $pos(\omega, \omega_r)$ and is defined as follows:*

- *if $\omega_r = \Lambda$, then $pos(\omega, \omega_r) = \omega$*
- *else, there exists $p \in \mathbb{Z}^*$ and $\omega'_r \in Rpos$ such that $\omega_r = p.\omega'_r$ and*

- *if* $p > 0$, *then* $pos(\omega, \omega_r) = pos(\omega.p, \omega'_r)$
- *if* $p < 0$ *and if there exists* ω' *and* ω'' *such that* $\omega = \omega'.\omega''$ *and* $|\omega''| = -p$, *then* $pos(\omega, \omega_r) = pos(\omega', \omega'_r)$

It is undefined everywhere else.

We note $t_{|\omega,\omega_r}$ *the term* $t_{|pos(\omega,\omega_r)}$ *for every* ω *and* ω_r *such that* $pos(\omega, \omega_r)$ *and* $t_{|pos(\omega,\omega_r)}$ *are defined. We name it the subterm of* t *referenced by* ω_r *from* ω.

Intuitively, ω_r describes a path back and forth inside t from ω to $t_{|\omega,\omega_r}$. For example, the relative positions -1.1 and $-2.1.2.-1.1$ reference the same subterm a of $f(s(a, b))$ from the position 1.2.

We can now define the notion of first-order terms with references. It only consists in extending an algebraic signature with an infinite set of constants denoting relative positions.

Definition 2.3 (Term with references) *For every set of first-order terms* $\mathcal{T}(\mathcal{F}, \mathcal{X})$, *the corresponding set of terms with references* $\mathcal{T}_{ref}(\mathcal{F}, \mathcal{X})$ *is the set* $\mathcal{T}(\mathcal{F} \cup Rpos, \mathcal{X})$ *where elements of Rpos have arity* 0.

As an example, $f(s(a, -1.1))$ is a term with references of $\mathcal{T}_{ref}(\{f, s, a\}, \emptyset)$. By abuse of notation, we will say that "-1.1 references a in $f(s(a, -1.1))$", without specifying it occurs at position 1.2.

Problems will inevitably occur when considering undefined relative positions. We define therefore validity as follows. We also forbid terms containing relative positions referencing relative positions.

Definition 2.4 (Term with references validity) *A term with references* $t \in \mathcal{T}_{ref}(\mathcal{F}, \mathcal{X})$ *is valid if for every leaf* $\omega_r = t_{|\omega}$ *such that* $\omega_r \in Rpos$, $t_{|\omega,\omega_r}$ *is defined and is not in Rpos.*

2.2 Implementation of terms with references

Let us now see how this formalism can be transposed to the Tom language. One characteristic of Tom is its data-structure independence. A term can be represented by any Java object as long as the user provides a mapping to see these objects as trees. For easier development, it comes up with a language called Gom [13] which automatically generates from a signature the Java implementation and the mapping. The resulting implementation is efficient in space and time (constant time terms equality test) because of maximal subterm sharing. Readers must pay attention to the difference between the maximal sharing and the notion of sharing in term-graphs. In our case, the maximal sharing is only at implementation level and does not lead to sharing at the term level. A Gom signature contains sorts and their constructors. For example, the signature below defines two sorts `A` and `B` along with their constructors.

```
A = a()
  | f(A)
  | s(A,A)
```

```
B = g(A)
```

With this signature, we can construct the terms `a()`, `f(a())` or `g(f(a()))` for instance. From this description, Gom generates a Java class hierarchy attaching an abstract class to each sort, and a class extending this sort class to each constructor.

Our goal is to generate an extended signature for terms with references from an initial Gom one. To achieve this, for every sort `T` of a Gom module, we generate a new constructor of rank `posT(int*)`. The notation `*` is the same as in [3, Section 2.1.6] and can be seen as a family of constructors with arities in $[0, \infty[$. The previous example is extended in this way:

```
A = a()                          B = g(A)
  | f(A)                           | posB(int*)
  | s(A,A)
  | posA(int*)
```

As an example, we can now build the extended term $s(-1.2.1, f(a))$ with the following syntax: `s(posA(-1,2,1),f(a()))`. Then `posA(-1,2,1)` references `a()` in the term `s(posA(-1,2,1),f(a()))`.

This type of terms with references using explicit relative positions constitutes a first extension of a Gom signature. In order to ensure type-preservation and reference correctness, a second representation level consists in expressing references with the help of labels. This notion of labelling can be seen as an implementation of the addressed terms presented in [4]. We have added new constructors to facilitate the use of labels and functions to transform a term with labels into the low-level representation. For every sort `T`, we generate two constructors. The constructor `labT(String,T)` enables the user to label a term with a string and `refT(String)` to reference a labelled term. Thus the term `s(refA("l"),f(labA("l",a())))` corresponds to the low-level term `s(posA(-1,2,1),f(a()))`. This notion of labels can be seen as syntactic sugar for hiding positions to users in order to avoid bad manipulations. Thereby, the constructors `posT` should be private so that users can only construct terms with references by label usage. We provide functions which generate the corresponding low-level terms after verifying that each `refT` corresponds to a `labT` of identical sort. This transformation is itself described using strategic rewriting introduced in 4.

2.3 Correspondence with term-graphs

Let us see now how a representation of term-graphs can be obtained from the terms with references introduced above. To ensure a bijection between term-graphs and their representations, we need to establish equivalence classes between terms with references. For example, $s(a, -1.1)$ and $s(-1.2, a)$ should be equivalent. They both correspond to the term-graph rooted by `s` whose two children correspond to the shared subterm a. Moreover, we noticed that several relative positions may reference the same subterm from a given position. Hence, we define canonical relative positions.

Definition 2.5 (Canonical relative position) *Let ω_1, ω_2 be two absolute positions, the canonical relative position $cpos(\omega_1, \omega_2)$ from ω_1 to ω_2 is the smallest relative position with respect to the length such that $pos(\omega_1, cpos(\omega_1, \omega_2)) = \omega_2$.*

Let us remark that $cpos(\omega_1, \omega_2) = q.\omega'$ where $\omega' \in (\mathbb{N}^*, .)$ and $q \in \mathbb{Z}^* \cup \{\Lambda\}$. We can now define the canonical form of terms with references using an order on absolute positions.

Definition 2.6 (Canonical term with references) *Let $\omega_1 = n_1.\omega_1'$ or Λ and $\omega_2 = n_2.\omega_2'$ or Λ be two absolute positions,*

$$\omega_1 <_\Omega \omega_2 \Leftrightarrow \begin{cases} & \omega_1 = \Lambda \\ or & n_1 < n_2 \\ or & n_1 = n_2 \text{ and } \omega_1' <_\Omega \omega_2' \end{cases}$$

A term t with references is then canonical if and only if t is valid and for every leaf $\omega_r = t_{|\omega}$ such that $\omega_r \in Rpos$, ω_r is canonical and $pos(\omega, \omega_r) <_\Omega \omega$.

Typically, contrary to $s(-1.2, a)$, the term $s(a, -1.1)$ is a canonical representation of a term-graph.

The formalism presented all along this section has been implemented through a plugin for Gom which generates an extended signature with new constructors for positions and construction functions which offer different levels of abstractions (from terms with explicit positions to term-graphs with labels). As illustrated by the Figure 1, a user may provide a labelled representation which is not a canonical form and use the provided construction function to normalize it. Whatever the favored level of the user,

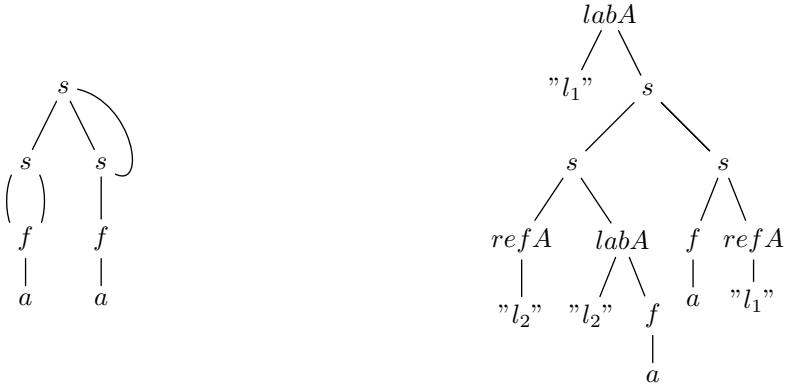

Fig. 1. An example of term-graph and its representation as a labelled term.

the in-memory representation is always based on explicit relative positions. Moreover, due to Gom design and in particular to the maximal sharing, the efficiency in time and space is ensured. For example, the term-graph presented Figure 1 is automatically translated during the construction into the low-level term with positions depicted in Figure 2. The principle of

maximal sharing is also illustrated by a schematic representation of the heap.

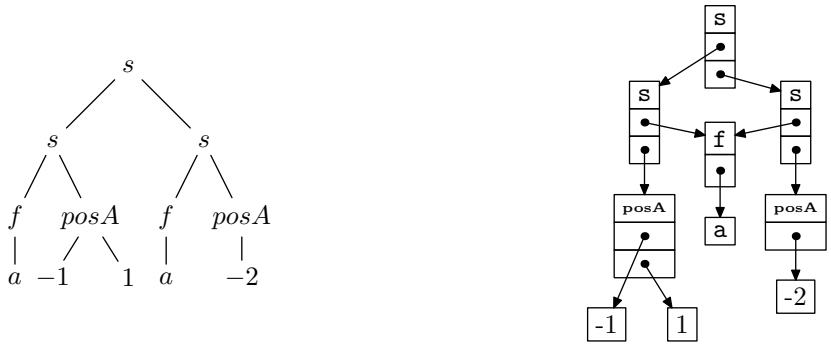

Fig. 2. Generation of relative positions from the labelled representation and maximal subterm sharing in memory.

After defining terms with references rewriting, we will exhibit in the next two sections how the Tom language offers strategic rewriting of these structures.

3 Term-graph matching

The originality of the previous approach is that pattern matching on terms with references build upon $\mathcal{T}(\mathcal{F}, \mathcal{X})$ is simply defined as pattern matching on terms of $\mathcal{T}_{ref}(\mathcal{F}, \mathcal{X})$. There is therefore no need to extend the notion of rewriting, which allows us to reuse existing results and rewriting tools. However, the questions raised by this formalism are situated at another level: we would like the rewrite system to rewrite only valid terms. Giving some non-trivial criterion on rewrite rules implying this property remains an open question for the moment. The next sections of this paper therefore focus on technical aspects of the pattern matching problem implementation.

After introducing the Tom language, we discuss various presentations of graph with references rewriting in this system. Although we cannot statically check that patterns ensure the validity of matched terms, we also propose several solutions to check this property at runtime.

3.1 Tom *pattern matching*

The first mechanism offered by the Tom language is pattern matching on algebraic terms. This feature is similar to the constructs proposed by functional languages like OCaml or Haskell. It is enabled by the `%match` keyword which allows us to match a subject against some pattern and to get the values of the pattern variables into Java ones:

```
A term = `s(f(a()),a());
%match(term) {
  s(x,y) ->  {
```

```
    System.out.println(
      "First child: " + `x + ", second child: " + `y
    );
    return `f(x);
  }
}
```

A subject is then any Java object which is an instance of a class whose description has been provided to Tom via a *mapping*. This mapping indicates to the Tom compiler how to match some class against a pattern, and how to create new algebraic terms implemented by this class via the ` construct. Here we are using the classes generated by Gom along with their mappings. Tom also supports associative matching, a.k.a. list matching, as well as anti-patterns [8] and non-linear matching.

Let us elaborate on the mapping mechanism. It provides an algebraic view of some Java object (*e.g.* seeing integers as Peano natural numbers, or seeing an XML tree as a term). It is divided into two parts: the *destructive* part and the *constructive* one. The destructive part is used by the matching algorithm and its main function is to describe how to query a term about its head symbol and how to get its n^{th} child. For instance, the mapping between integers and Peano naturals would be similar to the following schematic code:

```
is_zero(n)                { n == 0 }
is_successor(n)           { n > 0 }
get_successor_child(n)    { n - 1 }
```

On the other hand, the constructive part is used by the compiler to build an algebraic term. It usually consists in calling the constructor of the Java class implementing the term. Although our goal is to work as much as possible on top of classes and mappings generated by Gom, we will punctually adapt some mapping to our needs.

3.2 Matching terms with references

Given these language constructs and the terms described in Section 2.2, there are many ways to express matching against patterns with references. As for term construction, patterns can be expressed at low-level using directly positions or by a syntax based on labelling. In each case, it refers to a stated subterm whose position is well-known. To compare two references by value instead of references, we will introduce a `deref` operator in patterns implemented using Tom mappings.

The simplest way to handle Gom terms with references is to consider the extended signature and perform some standard pattern-matching on it. Since the `posT(int*)` constructors generate matchable terms, it is possible to write patterns where relative positions are explicitly given. As an example, the

term represented Figure 3 matches against the pattern `s(a(),pos(-1,1))`. Notice that this type of pattern denotes exactly the structure of the term: *e.g.* `s(pos(-1,2),a())` would not match the same term. This method allows us to match against any position, even those pointing to an upper term as shown Figure 4. This may still be relevant in case of a procedure carrying some

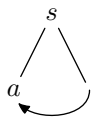

Fig. 3. `s(a(),pos(-1,1))`

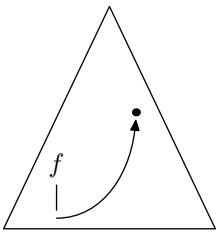

Fig. 4. `f(pos(-n,...))`

contextual information or fetching the position to perform some computation later. It may also be useful to compare two positions, without knowing the value of the subterms they are referencing. Figure 5 illustrates this situation. Notice however that this is only possible if the two variables have the same height in the term, as we are comparing *relative* positions.

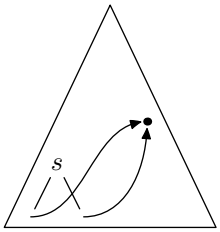

Fig. 5. `s(x,x)`

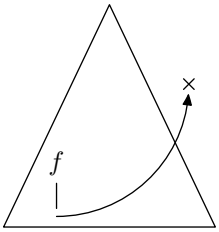

Fig. 6. `f(pos(-n,...))`

This solution presents two issues: the main one, depicted by Figure 6, is that a relative position may be undefined. These patterns should therefore be considered as a kind of unsafe assembly language for matching terms with references. The second one is that the explicit notation of positions is not mandatory and may be easily avoided with some syntactic sugar.

Thereby we propose to slightly modify the Tom compiler to address them. The first change consists in integrating labels capturing and denoting positions of subterms into the patterns syntax in order to avoid any explicit position matching. As an example, the term represented on Figure 3 would match against the pattern `s(x:a(),x)`. The translation of this kind of patterns to the former one is trivial: each occurrence of a label `lab` is replaced by the relative position from its position to the position of the subterm labelled by `lab`.

The second modification aims at reinforcing the patterns safety. As explained in section 2.2, we do not want the user to be able to recover a position by matching the term of figure 3 against `s(_,x)` for instance.

This can be achieved by inhibiting the generation of mappings for position constructors, so that the matching algorithm fails on such patterns. Another less restrictive way of dealing with the undefined relative positions problem would be to have the patterns similar to `s(_,x)` match only valid terms. This could be achieved by checking at runtime that every relative position in `x` references an accessible term. This is easily done with the help of strategies presented in section 4. In both cases, we cannot avoid some modification of the pattern-matching algorithm, thus of the compiler.

The two previous kinds of patterns focus on the positions themselves as matchable objects. Another approach would be to have the patterns express constraints about the value of the referenced subterms. The mapping mechanism presented in Section 3.1 offers the necessary features to achieve this via the writing of an *ad hoc* destructor. We wrote this `deref` destructor which acts like a proxy between the pattern matching algorithm and the destructor of the value referenced by a position. As an example, the term represented by

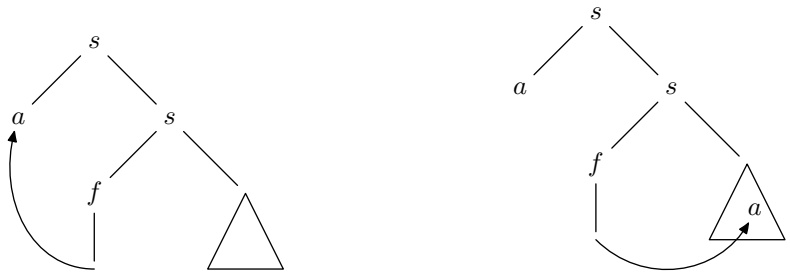

Fig. 7. `deref(a())` ambiguity

Figure 3 matches against the pattern `s(a(),deref(a()))`. It is important to note that the patterns are now an abstraction of the term so we do not match the graph *structure* anymore. For instance, the two terms of Figure 7 match against the same pattern `s(a,s(f(deref(a())),_))`. In particular, it is not possible anymore to use non-linear pattern matching in order to check that two positions are referencing the same sub-term, as depicted by Figure 8 which shows the ambiguity of the `s(s(deref(x),deref(x)),_)` pattern. Again, matching terms with references in this way is not safe. Indeed the

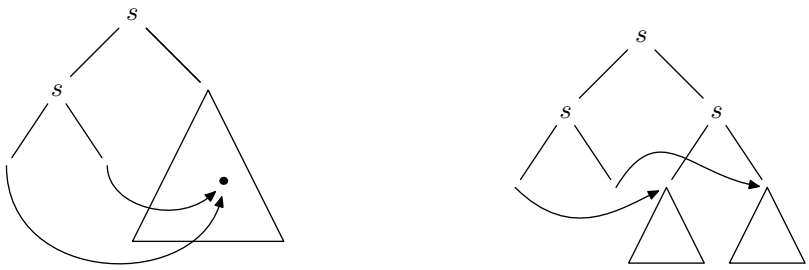

Fig. 8. `deref(x),deref(x)` ambiguity

subject may contain positions referencing terms above its root. However

this time, checking the validity of a term does not require any change to the compiler since the test can be transfered to the destructor. The later aborts the matching process by returning `false` if accessing the pointed term raises an exception.

3.3 Matching term-graphs

Contrary to Gom terms with references, the usual term-graph definition does not differentiate two types of children. Therefore, it may be convenient to have the patterns `s(x:a(),x)` and `s(x,x:a())` match either `s(a(),pos(-1,1))` or `s(pos(-1,2),a())`. The normal form mentioned in Section 2.2 enables such a feature: it is sufficient to maintain normalization of both terms at runtime and patterns at compile time to ensure this behavior. It requires some minor changes of the Tom compiler though.

As recalled in Section 2.2, one main application of term-graphs is the representation of subterms sharing in the purpose of gaining space and computation time. However, this structure (the sharing) does not reflect the structure of the represented term (typically a λ-term) and it is therefore desirable to manipulate it modulo this encoding. The basic idea is to interweave `deref` constructors inside the patterns, so that `s(a(),a())` is translated into `deref(s(deref(a()),deref(a())))` and thus matches the graph of figure 3. It only requires to confer some lazy behavior to the `deref` destructor, which should act as if not existing in case of a direct subterm (not a position).

Even if the classical [2] representation of term-graphs by a labelled graph is similar to ours, the conditions on rewrite rules are more restrictive (the left-hand side of a rule is limited to trees). For now, term-graph rewriting in Tom is expressed by syntactic term rewriting. Contrary to [2], there is no garbage collection phase and referenced subterms can disappear or change, leading to invalid terms. One solution would be to integrate this garbage collection phase in the Tom matching. An other attractive approach would be to implement the formalism presented in [5] where the right-hand side of the rewriting rules consists in a set of actions on the pointers.

4 Strategic programming with term-graphs

Tom provides a powerful strategy language inspired by ELAN and Stratego. The purpose of strategies is to describe how transformation rules should be applied. In case of terms with references, the strategy language must be extended in such a way that we can traverse them as graphs.

4.1 Tom strategy language

Elementary strategies are composed of the two basic strategies `Identity()` and `Fail()` as well as type-preserving user-defined rewrite rules specializing

their behaviour:

```
%strategy Eval() extends Fail() {
  visit A {
    s(x,a()) -> { return `f(x); }
    s(x,y)  -> { return `y; }
  }
}
```

When applied to a node of sort `A`, a transformation is performed if one of the patterns matches the node. Otherwise, the default `Fail` strategy is applied.

More complex strategies can be built on top of elementary ones, involving basic combinators introduced in ELAN [7] and extended in [14]: `Sequence(s1,s2)`, `Choice(s1,s2)`, `All(s)`, `One(s)`, *etc.* We can therefore build strategies such as `` `Choice(Eval(),Identity()) `` which tries to apply `Eval()` to the current node and returns it unchanged if `Eval()` failed (*i.e.* none of the patterns matched the current node).

Besides, the strategy language allows the declaration of recursive parametrized strategies, enabling the definition of higher-level constructs. For example, the fix-point operator can be expressed by $\texttt{Repeat(s)} \triangleq \mu\texttt{x.Choice(Sequence(s,x), Identity())}$, where μ denotes a recursion operator, `x` a variable, and `s` a parameter of the strategy. In Tom, we raised the recursion operator to the object level, allowing the definition of complex strategies in a truly algebraic manner:

```
Strategy Repeat(Strategy v) {
  return `mu(MuVar("x"),
             Choice(Sequence(v,MuVar("x")),Identity()));
}
```

Finally, Gom generates a congruence strategy `_f` for each constructor `f` of an algebraic signature. Using the notation `s[t]` to express the application of the strategy `s` to the term `t`, $\texttt{\_f(s}_1\texttt{,...,s}_n\texttt{)[f(c}_1\texttt{,...,c}_n\texttt{)]}$ returns $\texttt{f(s}_1\texttt{[c}_1\texttt{],...,s}_n\texttt{[c}_n\texttt{])}$ and fails if the head symbol of the subject is not `f`. This allows to perform pattern matching "on the fly" during term traversal.

One noticeable property of strategic programming with Tom is that it is possible to get the current absolute position inside the visited term during a traversal. This allows for instance to collect in one pass the set of reduced forms of a term for a given rewrite system. In our case, we will make use of this feature in the next section to collect the positions of bounded variables occurences under an abstraction.

4.2 Extension of Tom strategy language

In order to traverse terms with references, we enrich the strategy language of Tom with one new strategy combinator `Ref` whose semantics is defined as

follows:

$$\texttt{Ref(s)[t]} = \begin{cases} \texttt{s[t']} \text{ if } \texttt{t'} \text{ is the term referenced by } \texttt{t} \\ \texttt{s[t]} \text{ otherwise} \end{cases}$$

This new basic combinator can be used everywhere in a composed strategy. One important characteristic of the Tom strategy language is that every composed strategy is itself a term and therefore can be traversed and rewritten. Adapting a strategy term for graphs with references consists in weaving the `Ref` combinator ahead every elementary strategy inside a strategy term. For example, `Sequence(s1,s2)` where `s1` and `s2` are elementary strategies will be rewritten into `Sequence(Ref(s1),Ref(s2))`.

5 Application to the lambda-calculus

Let us see now some application of our programming framework through the implementation of a basic λ-calculus interpreter. The graph with references will encode variable bindings, acting as de Bruijn indices, while the strategy language will translate the usual evaluation strategies of λ-calculus.

We work with a minimalist Gom signature:

```
LT = App(LT, LT)
   | Abs(LT)
```

The chosen representation of λ-terms makes use of terms with references by replacing variables with positions pointing to the corresponding binder. For instance, the term $\lambda f.\lambda x.(f\, x)$ will be encoded by the Gom term `Abs(Abs(App(posLT(-3),posLT(-2))))`. This encodes a kind of de Bruijn indices counting not only abstractions but also every node in the syntactic tree of the λ-term.

Let us write a `beta` strategy wich performs one β-reduction step on a redex. As mentioned in the previous section, it is possible to get the current position inside a visited term during its traversal by a strategy. Thereby, knowing the position of λ inside the visited redex $(\lambda x.f\ a)$ will allow us to find all the occurences of x in f, *i.e.* relative positions pointing to λ. The `beta` strategy then simply consists in four steps when applied to an application $(\lambda x.f\ a)$:

(i) collecting the position of λ;

(ii) collecting a;

(iii) replacing all the occurences of relative positions pointing to λ by a in f;

(iv) replacing the redex by the modified f.

Assuming we have a mutable structure `info` (a Java class here) which can store both informations of the first and second steps, this is achieved by the

following strategy:

```
Strategy beta = ‘Sequence(
        _App(Identity(),collectTerm(info)),
        _App(
            Sequence(
                collectPosition(info),
                _Abs(μx.Choice(substitute(info),All(x)))),
            Identity()),
        clean());
```

We can notice the presence of four user defined strategies: `collectTerm`, `collectPosition`, `substitute` and `clean`. They respectively perform the four steps described above. Their code is obvious and one line long, except for the `substitute` strategy which has to compute the absolute position referenced by the current term to compare it with the position of λ stored in `info`. Then it performs the necessary shifts on bounded variables (relative positions) inside a before returning it. The whole strategy itself is an overlapping of congruence strategies. The `μx.Choice(substitute(info),All(x))` construct means that we do not go down further inside the term if the substitution succeeded.

We shall now apply this `beta` strategy on a λ-term with some evaluation strategy until we reach a fixpoint. `beta` being a strategy, it can be combined with other strategies to perform reductions. In particular, the `TopDown` and `Innermost` strategies respectively encode call-by-name and call-by-value evaluation strategies modulo some fixpoint computation encoded by the provided `RepeatId` strategy. They are themselve expressed using elementary strategies:

```
  TopDown(s) = μx.Sequence(s,All(x));
Innermost(s) = μx.Sequence(AllRL(MuVar(x)),Try(Sequence(s,x)))
```

Where `AllRL` applies `s` to all the childs of the current node from right to left. Substituting `s` by `beta` inside one these enables the expected evaluation behaviour.

Let us finally see how a typical use of term-graphs, namely subterms shared evaluation, can be implemented by a slight modification of the previous example. We now assume that many bounded variables are represented by shared subterms where “shared” is meant in the sense of term-graphs semantics. For example, the λ-term $\lambda x.(x\,x)$ will be represented by `Abs(App(posLT(-2),posLT(-1,1)))` instead of `Abs(App(posLT(-2),posLT(-2)))`. The previous `beta` strategy is then still mainly valid since this modification only affect the situations where the second child of an application is a variable, *i.e.* a relative position. Hence, changing the line `_App(Identity(),collectTerm(info))` by

`_App(Identity(),Ref(collectTerm(info)))` suffices to adapt the strategy to the new λ-terms representation. This modification is of course relevant in case of a call-by-name strategy.

The discussed implementation is available in the Tom subversion repository [1], under the `examples/termgraph` path.

6 Conclusion

To the best of our knowledge, we have presented here a new way of representing terms with references which presents strong similarities with the term-graph formalism. Using the Tom language as a programming background, we have discussed the various advantages and drawbacks of such an approach at different levels: memory representation, pattern matching and strategic traversal. We finally presented an application of this framework via the writing of a simple λ-calculus interpreter making an heavy use of strategies.

Although a major part of the presented propositions has been implemented, the solutions requiring a modification of the compiler as described in section 3.2 are still theoretical. The syntactic sugar constituted by the labelled notation is a matter of few lines of code. On the other hand it may be interesting to look closely at an efficient way of checking the validity of references contained by a given subterm at runtime. Another field of investigation would be the writing of `Ref` strategies aborting infinite loops appearing during the traversal of a graph with cycles. This could be achieved by some map associating counters to visited nodes.

As shown by the last section, this model has interesting applications and opens promising perspectives in terms of program transformation and code analysis. Besides, the normal form described in section 2.2 makes it a solid basis for experimenting transformations on term-graphs in a concise and expressive manner.

References

[1] Ariola, Z. M. and J. W. Klop, *Equational term graph rewriting*, Fundam. Inf. **26** (1996), pp. 207–240.

[2] Barendregt, H. P., M. van Eekelen, J. Glauert, J. Kennaway, M. Plasmeijer and M. Sleep, *Term graph rewriting*, in: *PARLE Parallel Architectures and Languages Europe*, Lecture Notes in Computer Science **259** (1987), pp. 141–158.

[1] Compilation instructions are detailed in the Tom documentation at `http://tom.loria.fr/docs.php`

[3] Comon, H. and J.-P. Jouannaud, *Les termes en logique et en programmation* (2003), master lectures at Univ. Paris Sud.
URL http://www.lix.polytechnique.fr/Labo/Jean-Pierre.Jouannaud/articles/cours-tlpo.pdf

[4] Dougherty, D. J., P. Lescanne and L. Liquori, *Addressed term rewriting systems: Application to a typed object calculus*, Mathematical Structures in Computer Science **16** (2006), pp. 667–709.

[5] Echahed, R. and N. Peltier, *Narrowing data-structures with pointers.*, in: *ICGT*, 2006, pp. 92–106.

[6] Kennaway, R., *On graph rewritings*, Theoretical Computer Science **52** (1987), pp. 37–58.

[7] Kirchner, C., H. Kirchner and M. Vittek, *Designing constraint logic programming languages using computational systems*, in: F. Orejas, editor, *Proceedings of the 2nd CCL Workshop, La Escala (Spain)*, 1993.

[8] Kirchner, C., R. Kopetz and P. Moreau, *Anti-pattern matching*, in: *Proceedings of the 16th European Symposium on Programming*, 2007.

[9] Lacey, D. and O. de Moor, *Imperative program transformation by rewriting*, in: *Proceedings of the 10th International Conference on Compiler Construction* (2001), pp. 52–68.

[10] Löwe, M., *Algebraic approach to single-pushout graph transformation*, Theoretical Computer Science **109** (1993), pp. 181–224.

[11] Moreau, P.-E., C. Ringeissen and M. Vittek, *A Pattern Matching Compiler for Multiple Target Languages*, in: G. Hedin, editor, *Proceedings of the 12th International Conference on Compiler Construction*, LNCS **2622** (2003), pp. 61–76.

[12] Plump, D., "Handbook of Graph Grammars and Computing by Graph Transformation," World Scientific Publishing, 1999 pp. 3–61.

[13] Reilles, A., *Canonical abstract syntax trees*, in: *Proceedings of the 6th International Workshop on Rewriting Logic and its Applications*, 2006, to appear.

[14] Visser, E. and Z.-e.-A. Benaissa, *A core language for rewriting*, in: C. Kirchner and H. Kirchner, editors, *Second International Workshop on Rewriting Logic and its Applications*, Electronic Notes in Theoretical Computer Science **15** (1998).